Declaration-level static assertions

WG14 N3641

Céleste Ornato <celeste@ornato.com>

July 6, 2025

- **Proposal category**: Feature request
- **Target audience**: Developers writing libraries

1 Abstract

This proposal aims to add the `[[assume(expr)]]` and `[[assume(expr, msg)]]` attributes to function declarations, allowing the designer of a library to specify statically-verifiable constraints in function parameters.

This features allows both for added safety on the caller side, and possibly for further optimisation on the compiler side.

2 Foreword

I have, between the initial writing of this paper and its submission, been made aware of the fact that C++26 has implemented a similar feature, through "function contract specifiers".

While similar in the abstract, the proposed feature is fairly different in practice: For one, It aims to add compile-time security, rather than at execution-time. It also does not add any new syntax, insofar as functions compiled using this feature should still be callable from older code without any issue[1].

I believe it is still important to define this feature independently from other standards, so as to make something that suits the needs and aims of C developers as well as possible.

[1] This is only guaranteed if the caller adheres to the guidelines set by the library developer regarding proper usage of the function, see 4.2.

3 Introduction

Inattention and forgetfulness cause developers to write insecure code.

If one wishes to convey to the user of a library that certain function parameters can cause undefined or unwanted behaviour, they can only do so in written documentation. This leaves the developer in a precarious position: they can either add checks at execution-time, which come with an unwanted performance overhead and having to setup and document an error code system, or they can assume that everyone will read the documentation, which may create vulnerabilities.

This proposal seeks to make some of these checks possible without overhead, by implicitly making static assertions before the function call.

This feature can be seen as an extension to the C99 "static array index in function parameters" feature, in that it both allows for optimisations by the compiler and for more compile-time checking for the user.

4 Proposal

4.1 Technical Description

When the attribute `[[assume(expr)]]` is associated to the declaration of a function `foo`, any call to `foo` will require the compiler to check that `expr` is **not provably false** (see 4.4) given the parameters. When compiling `foo`, it is assumed that `expr` is always true. Calling the function with parameters that do not respect `expr` is undefined behaviour.

`expr` may address the function parameters by their name, or by calling them `$n`, where `n` is the 0-indexed number of the parameter from left to right.

If a referenced variable is an array, it should decay into a pointer. While it could have allowed for more features, this would have created bug-prone behaviour depending on whether the author of `expr` expected a pointer or an array. In any case, most problems related to the size of arrays are already solved by having static indices in the function signature.

4.2 Rationale

Adding more undefined behaviour in C2y might be controversial, it would certainly not be immediately seen as "Enabling secure programming".

This feature is meant for cases where the developer already considers certain parameters to be "Undefined behaviour". At this point, it does not matter whether the standard considers the code to be defined, because the results would still be unexpected and prone to breakage.

Allowing for further optimisation is just a welcome consequence of one being able to specify what they consider undefined behaviour. In reality, the wanted feature is the compiler being capable of detecting mistakes. Function signatures using this attribute also make the code self-documenting, which makes it easier to understand the assumptions made by the developer when writing the function.

4.3 Example

```
[[assume($1 != 0)]]
int division(int, int);

[[assume(a >= b, "Substraction requires a >= b")]]
unsigned int substract(unsigned int a, unsigned int b);

int main(void)
{
    // These compile, with no execution-time overhead.
    int result1 = division(1, 2);
    unsigned int result2 = substract(2, 3);

    // Error: "Assumption '$1 != 0' is false."
    int result3 = division(2, 0);
    // Error: "Substraction requires a >= b."
    unsigned int result4 = substract(0, 6);
}
```

4.4 Quirks

An assumption whose validity cannot be proven will be treated as valid. This should not be a problem, as this would just mean coming back to the status quo of having to be careful as a user. This is consistent with the way in which static array indices work in function signatures.

Due to the way static assertions work in general, we cannot always assert the validity of every expression. However, in "safer", more restricted styles of C, where we only work with automatic-storage-duration variables and constant-sized arrays, this feature proves itself quite powerful already.

With C23 came more ways to define constant data, through `constexpr`. Assuming it ever reaches parity with C++ `constexpr`, or considering the possible presence of N3600 in C2y, this feature could continue to improve over time.

5 Implementation

Seeing as the more complex C++ "contract" system will in any case be implemented by C++26-compliant compilers, this feature should not pose a problem to C2y-compliant compilers. In any case, as was previously stated, calling a function compiled with `assume` would still work on previous standards in library-compliant contexts.

There is seemingly no C compiler extension allowing for this exact feature. One could imagine possible function-like macros being able to replicate such a feature, but it would certainly be non-trivial.

Even then, macros would not be ideal, as they would:

1. Allow for the library user to call the function without its underlying assumptions,
2. Make compile-time optimisations impossible,
3. Clutter the program with extraneous definitions if we have one macro per function,
4. Be incompatible with styleguides wherein parameters are unnamed in declarations,
5. Generally worsen the user experience, as macros are not always well supported by language servers,
6. Make the assumptions messy and hard to modify; and
7. Come with the usual points of failure of macros (CERT-PRE31-C, notably).

Indeed, it would be much more interesting for it to be a standard feature instead of being bound by the rules of macros.